Unifying Single-Agent and Two-Player Search

Jonathan Schaeffer
Computing Science Dept.
University of Alberta
Edmonton, Alberta
Canada T6G 2H1
jonathan@cs.ualberta.ca

Aske Plaat
Computer Science Dept.
Vrije Universiteit
Amsterdam
The Netherlands
aske@xs4all.nl

Andreas Junghanns
Research and Technology
DaimlerChrysler
Berlin
Germany
andreas.junghanns@daimlerchrysler.com

Abstract

The seminal works of Nilsson and Pearl in the 1970's and early 1980's provide a formal basis for splitting the field of heuristic search into two subfields: single-agent and two-agent search. The subfields are studied in relative isolation from each other; each having its own distinct character. Despite the separation, a close inspection of the research shows that the two areas have actually been converging. This paper argues that the single/two-agent distinction is no longer of central importance for heuristic search anymore. The state space is characterized by a number of key properties that are defined by the application; single-agent versus two-agent is just one of many. Both subfields have developed many search enhancements; they are shown to be surprisingly similar and general. Given their importance for creating high-performance search applications, it is these enhancements that form the essence of our field. Focusing on their generality emphasizes the opportunity for reuse of the enhancements, allows the field of heuristic search to be redefined as a single unified field, and points the way towards a modern theory of search based on the taxonomy proposed here.

1 Introduction

Heuristic search is one of the oldest fields in artificial intelligence. Nilsson and Pearl wrote the classic introductions to the field [34, 36]. In these works (and others) search algorithms are typically classified by the kind of problem space they explore. Two classes of problem spaces are identified: state spaces and problem reduction spaces. Many problems can be conveniently represented as a state space; these are typically

problems that seek a path from the root to the goal state. Other problems are a more natural fit for problem reduction spaces, typically problems whose solution is a strategy. Sometimes both representations are viable options. Problem reduction spaces are AND/OR graphs; AO* is the best-known framework for creating search algorithms for this class of problems [3, 34]. State spaces are OR graphs; the A* algorithm can find optimal solutions to this class of problems [17]. Note that a state space (OR graph) is technically just a special case of a problem reduction space (AND/OR graph).

Since their inception, the notions of OR graphs and AND/OR graphs have found widespread use in artificial intelligence and operations research. Both areas have active research communities which continue to evolve and refine new search algorithms and enhancements. Of the two representations, the state space representation has proven to be the more popular. It appears that many real-world problem solving tasks can be modeled naturally as OR graphs. Well-known examples include the shortest path problems, sliding-tile puzzles, and NP-complete problems (such as the traveling salesperson, bin packing and job-shop scheduling).

One application domain that fits the AND/OR graph model better is two-agent (two-player) games such as chess. In these games, one player chooses moves to maximize a payoff function (the chance to win) while the opponent chooses moves to minimize it. Thus, the AND/OR graphs become MIN/MAX graphs, and the algorithms to search these spaces are known as minimax algorithms. Curiously, it appears that two-player games are the *only* applications for which AND/OR algorithms have found widespread use. To contrast A*-like OR graph algorithms with two-player minimax algorithms, they are often referred to as single-agent (or one-player) search algorithms.

With the advent of Nilsson's AND/OR framework, two-agent search has been given a firm place within the larger field of heuristic search. Since AND/OR graphs subsume OR graphs, there is a satisfying conceptual unification of the two subfields. However, the impact of this unified view on the practice of research into heuristic search methods has been minor. The two subfields have continued to develop in parallel, with little interaction between them. Few scientists have studied both areas.

This article has the following contributions:

- Single-agent and two-agent search algorithms both traverse search graphs. The difference between the two algorithms is not in the graph, but in the semantics imposed by the application. Much of the research done in single-agent and two-agent search does not depend on the search algorithm, but on the properties of the search space. In fact, by matching basic graph traversal algorithms from both fields, it is shown how similar, if not identical, single-agent and two-agent search really are.

- Nilsson's [34] and Pearl's [36] dichotomy—the OR versus AND/OR choice—is misleading. Heuristic search consists of identifying properties of the search space and implementing a number of search techniques that make effective use of these properties. There are many such properties, and the choice of backup rule (minimaxing in two-agent search; maximization or minimization in single-agent search) is but one. The implication of Nilsson's and Pearl's model is that the choice of backup rule is in some way fundamental; it is not. This paper argues for viewing heuristic search as the process in which properties of a search space

are specified. Once that has been done, the relevant search techniques (basic algorithm and enhancements) follow naturally.

- Over the years researchers have uncovered an impressive array of search enhancements that can have a dramatic effect on search efficiency. The typical scenario is that the idea is developed in one of the domains and possibly later reinvented in the other. In this paper we list search-space properties under which many search enhancements are applicable, showing that the distinction between single-agent and two-agent search is not essential. By merging the work done in these two areas, the commonalities and differences can be identified. This provides the basis for constructing a generic search framework for designing high performance search algorithms — given the properties of the domain, an appropriate set of search enhancements can be automatically selected for consideration.

The message of this article is that single-agent and two-agent search can and should be considered as a single undivided field. It can, because the essence of search is *enhancements*, not algorithms as is usually thought. It should, because researchers can benefit by taking advantage of work done in a related field, without reinventing the technology, if they would only realize its applicability. Given all the similarities between the two areas, one has to ask the question: why is it so important to make a distinction based on the backup rule?

Some might object to the preceding discussion, arguing that two-player trees are searched differently than single-agent trees. In most two-agent search applications (such as game-playing programs), the search result is based on a depth-limited search; no goal is found. In contrast, single-agent searches have the objective of finding a goal state. The traditional view then is that two-agent search is looking for the best answer given a time constraint (satisficing) while single-agent search ignores the time constraints and searches for the best answer (optimality). While this categorization might be an accurate reflection of research done in these areas, it is only a generalization. Consider two counter examples. First, RTA* (Real-Time A*) was developed to search single-agent trees under tight resource (time) constraints [25]. Since the algorithm usually cannot find a goal state within its resource constraints, it approximates an answer by exploring a limited amount of the search space, just like stereotypical two-agent search programs do. Second, when solving two-player games, such as Tic-Tac-Toe or Nine Mens Morris, the aim is to find a goal node whose value can be propagated to the root of the search tree. Here the objective is to find an optimal result, not a satisficing one. These examples illustrate that the traditional view of single-agent and two-agent search is but a generalization, and that the characteristic highlighted here (optimality versus satisficing) is really just a user-defined constraint on the quality of the result. It is not fundamental to the algorithms.

In effect, single-agent and two-agent search (as well as other AI search algorithms) are essentially just graph traversal algorithms. There are only a few basic classes of ideas for efficiently searching graphs. It makes more sense to classify search implementations on the techniques used to efficiently search the graph, rather than the backup-rule semantics imposed by the application. An alternative way of looking at

this is to classify applications by the search-space properties that allow certain search enhancements to work.

A remark on how the concepts *algorithm*, *enhancement*, and *search technique* are used. Search algorithms are the basic graph traversal mechanisms as found in the text books, such as Nilsson's and Pearl's. They form the skeleton to which the human problem solver adds enhancements to achieve performance improvements. The term search techniques is used in this article to indicate enhancements and algorithms together. As this article advances the view that the basic algorithm decisions are easy and that the essence lies in the enhancements, the term search technique will usually be synonymous with search algorithm enhancement.

This article is organized as follows: Section 2 discusses the importance of search enhancements. Section 3 pairs up most of the major search algorithms from both single-agent and two-agent search. Section 4 gives a taxonomy of properties of the search space as described in Section 5, which are matched up with the applicable search techniques in Section 6. Section 7 puts this work in perspective. Section 8 draws some conclusions.

The article is restricted to classical heuristic search (single-agent and two-agent search) although the ideas are applicable to other graph-based search algorithms.

2 Algorithms vs Enhancements

Most introductory texts on artificial intelligence (AI) start off explaining heuristic search by differentiating between different search strategies, such as depth-first, breadth-first, and best-first. Single-agent search is introduced, perhaps illustrated using a sliding-tile puzzle. Another section is then devoted to two-player search algorithms. The minimax principle is explained, often followed by alpha-beta pruning. The focus in these texts is on explaining the basic search algorithms and possibly their fundamental differences (the backup rule and the decision as to which node to expand next). And that is where most AI books stop their technical discussion.

In contrast, in real-world AI applications, it is the next step—the search enhancements—that is the topic of interest, not so much the basic algorithm. The algorithm decision is usually easily made. However, the choice of algorithm enhancements can have a dramatic effect on the efficiency of the search. Although it goes too far to say that the underlying algorithm is of no importance at all, it is fair to say that most research and development effort for new search methods and applications is spent with the enhancements.

A search enhancement relies on the presence of specific search-space properties to improve the efficiency of the underlying algorithm, with respect to (a number of) resources. The presence of these properties suggests that the enhancement is applicable and is likely to be beneficial. Some of the enhancements are based on application-specific properties; others work over a wide range of applications all sharing the same property. Examples of application-dependent enhancements include the Manhattan distance for the sliding-tile puzzle, and first searching moves that capture a piece before considering non-capture moves in chess. Examples of application-independent

enhancements are iterative deepening [42][1] and cycle detection [16, 44].

Consider the example of the transposition table enhancement. Transposition tables rely on the search space being a graph. If the search space is a tree, transposition tables yield no direct benefits.[2] The basic idea, caching previously computed information, is obviously independent of the single-agent and two-agent distinction and its applicability only depends on a property of the underlying graph (is it a tree or not). Note also that the transposition table is often referred to as a hash table; we do not use the latter terminology since this refers to the details of an implementation.

The performance gap between search algorithms with and without enhancements can be large. Something as simple as removing repeated states from the search can lead to large reductions in the search tree (e.g. [44] using IDA* in sliding-tile puzzles; [40] using alpha-beta in chess). Combinations of enhancements can lead to reductions of several orders of magnitude. For example, in chess the effectiveness of alpha-beta pruning really depends on adding the right combination of enhancements [40]. In single-agent search the same has been found, for example, for sliding-tile puzzles [11] and Sokoban [20]. In contrast, the choice of search algorithm has often a relatively small effect. For example, in two-player games, many variations on alpha-beta have been proposed, but in the end their performance is, at best, a small improvement [37].[3]

In the traditional view, new applications are carefully analyzed until an appropriate algorithm and collection of algorithm enhancements is found that satisfies the user's expectations. In this view, each problem has its own unique algorithmic solution; a rather segmented view. In reality, most search enhancements are small variations of general ideas. Their applicability depends on the properties of the search space. Therefore, the basic idea behind most search enhancements are generally applicable to both single-agent and two-agent search. It is the search enhancements that tie single/two-agent search together, achieving the unity that Nilsson's and Pearl's models strived for, albeit of a different kind.

Focusing on properties of the search space identifies the differences that one should take into account when designing a high-performance searcher for a specific application. A single search framework can help to identify these differences. We should realize that heuristic search is usually not about different algorithms, but is mostly about putting search techniques together like LEGO blocks, exploiting the properties of the search space. In effect we should stress the things that are the same, not the differences.

[1] It is important here to point out that even though an implementation might differ substantially to accommodate a search enhancement, it is the difference in the search traversal that differentiates an algorithm from an enhancement. Therefore we feel justified in calling iterative deepening an enhancement, even though the implementation differences between, for example, A* and IDA* are significant.

[2] The transposition table is routinely used to store other information that can benefit the search, such as move ordering hints. We regard these kinds of benefits as "indirect".

[3] This holds true for other minimax search algorithms, such as B* [8] and Conspiracy Numbers [30], where attempts to improve on alpha-beta have been unsuccessful. These innovative algorithms differ from alpha-beta in their leaf node values and the backup rule. However, they are still graph-traversal algorithms and the same set of search enhancement principles apply.

3 Search Algorithms

In this section we contrast many of the single-agent and two-agent search algorithms. If there is anything to the claim that the two fields of search should be unified because the only (artificial) difference is the backup rule, then a comparison of the algorithms used in the two fields should yield more similarities than differences.

Naive (depth-first single-agent search; no depth restriction minimax search). Both algorithms traverse a tree in a depth-first, left-to-right manner. Recursion stops when a terminal node is reached, without any depth restriction. The only application-dependent knowledge used is for assigning values to terminal nodes and choosing the correct backup rule. The storage requirements are proportional to the depth of the tree.

Simple (branch-and-bound; no depth restriction alpha-beta). These algorithms also search a tree depth-first and left-to-right without any depth restriction. They differ from their naive counterparts in that both use partial results obtained during the search to cut off parts of the tree that are irrelevant to the solution. Note that no additional application-dependent knowledge is needed here. The storage requirements are proportional to the depth of the tree.

Breadth (breadth-first single-agent search; iterative deepening minimax). New nodes are expanded one depth at a time. Both algorithms can use storage that is proportional to the depth of the tree by using depth-limited depth-first search. If sufficient storage is available, these algorithms can store the search tree in memory, potentially eliminating the repeated node visits that depth-first search entails. Note that single-agent search algorithms can get by with less storage, since they need only save the frontier of the search (since the search value is the minimum or maximum of the nodes); two-agent search requires access to the previous iteration's nodes (because it has to build a proof tree).

Informed (A*; SSS*). With the availability of heuristic evaluations, the notion of a "best" or "most promising" node to expand can be defined. A* and SSS* expand any one of the "best" nodes next. In both algorithms, a sorted OPEN-list keeps track of the nodes at the frontier and allows for expanding the best next node. Although conceptually A* and SSS* are similar algorithms, it is interesting to note that SSS* is equivalent to a depth-first search variant of alpha-beta (MTD(f) [37]).

Space efficient informed (iterative deepening A*; iterative deepening alpha-beta). A* and SSS* both suffer from exponential space requirements. Iterative deepening turns traditional list-driven best-first searches into multiple depth-first searches that increase the "depth" limit with each iteration. "Depth" should not be interpreted literally; it is synonymous with any useful criteria that enables a guaranteed cutoff of each branch in the search.

Real-time informed search (RTA*; depth-limited iterative deepening alpha-beta). When given resource constraints do not allow for the complete traversal of the search

space to find a solution, a (depth or time) limit is introduced. This partial search-space traversal backs up heuristic values about where a solution is likely to be. When resources run out, a ”best” move decision is made based on this information.

The above list is meant to be illustrative, not comprehensive. The comparison shows that the basic graph traversal approaches are similar, if not identical, between the “different” fields. The major perceived difference between single-agent and two-agent search is in the choice of algorithm that is commonly used. Single-agent applications usually optimize and hence use an informed (or space efficient informed) algorithm. Two-agent applications commonly satisfice and typically use a real-time informed search. However, their respective counterparts exist and are used.

4 Search Application Development

Two distinct issues play a role in the process of designing a high-performance search application: the properties of the state space, and the search algorithms and techniques used to find the desired information in that space. Therefore, search program design consists of two parts. First, the problem solver must specify the properties of the problem space. Second, based on this information, an appropriate implementation is chosen.

- Phase 1: *Search Space Definition.* The characteristics of the problem space must be specified.
    1. *Graph Definition:* The problem definition allows one to construct a graph, where nodes represent states, and edges are state transition operators. This is typically just a translation of the transition rules to a more formal (graph) language. It provides the syntax of the state space.
    2. *Solution Definition:* Goal nodes are defined and given their correct value. A rule for combining the values of a node’s successors to determine the value of the parent node is provided (such as minimization, or minimaxing). This adds semantics to the state space graph.
    3. *Resource Constraints:* Identify execution constraints that the search algorithm must conform to.
    4. *Search Objectives:* The problem solver defines the goal of the search: an optimal or satisficing answer (the quality of the answer).
    5. *Domain Knowledge:* Non-goal nodes may be assigned a heuristic value (such as a lower bound estimator or an evaluation score). The properties of the evaluation function fundamentally influence the effectiveness of many search enhancements, typically causing many iterations of the design-and-test cycle.
- Phase 2: *Program Design and Implementation.* Once the search space is specified, the problem solver can design the application program. The design process consists of three steps:

1. *Search Algorithm:* Typically the choice of basic algorithm is easily made based on the problem definition. Nilsson's and Pearl's model addresses the basic graph traversal questions concerning the backup rule and breadth-first/depth-first/best-first search strategy, based on the graph definition and the performance specifications. The single/two-agent distinction is usually unambiguous (dictating the backup rule to use), and the algorithm selection is often trivial.
2. *Search Enhancements:* The literature contains a host of search enhancements to exploit specific properties of the search space. The right combination can dramatically improve the efficiency of the basic algorithm. Textbook algorithms have to be revised substantially to accommodate inclusion of common enhancements. Note that an enhancement can even simplify the basic algorithm. For example, compare the code of iterative deepening A* to the original A*.
3. *Implementation Choices:* Given a search enhancement, the best implementation is likely to be dependent on the application and the choice of heuristics. For most applications, the majority of the design effort involves judiciously fine tuning the set of algorithm enhancements [20, 22]. For these choices we rely on the programmer's discretion (trading off programming effort for better search performance), which is beyond the scope of this paper.

The applicability of search algorithm enhancements is determined by the five categories of state space properties given above. Figure 1 summarizes the interaction between the state space properties (vertical axis) and step (b) of the algorithm design process—the search enhancements (the horizontal axis). A sample of enhancements are illustrated in the figure. The table shows how the search enhancements match up with the properties. An "x" means that the state space property affects the effectiveness of the search enhancement. A "v" means that the search enhancement (favorably) affects a certain property of the search space. For example, the "v"s on the row for time constraints indicate that most search enhancements make the search go faster. Star "*" entries mean that a search enhancement was specifically invented to exploit a property.

A detailed description of the search-space properties (vertical axis in Figure 1) follows in the next section. Each of the enhancements (horizontal axis in Figure 1, and others), as well as how they relate to the search space properties, are explained in Section 6.

5 Search Space Properties

This section gives details of the fundamental search-space properties that search enhancements attempt to exploit (the vertical axis in Figure 1).

Search Space Properties \ Enhancements	Cycles detection	Exhaustive search	Move ordering	Iterative deepening	Solution databases	Static redistribution of search effort
Graph Definition						
out degree of a node			> 1			> 1
in degree of a node	> 1					
presence of cycles	x					
graph size		x				
Solution Definition						
solution density		x			x	
solution depth					x	
solution backup rule		x			x	
Resource Constraints						
space	x	x		*	x	
time	v	v	v	*/x	v	v
Search Objectives						
optimization	v	v	v	v	v	
satisficing	v		v	v	v	v
Domain Knowledge						
heuristic evaluation quality			x	x	*	x
heuristic backup rule						
heuristic parent/child value			x	x		x
heuristic parent/child state			x			x
heuristic evaluation granularity				x		
next move to expand	*	v	*	v		*

Figure 1: Search Space Properties vs Enhancements

5.1 Graph Definition

The first part of defining a search space is the problem specification phase. Here the problem has to be formulated in an unambiguous way, essentially defining a problem space that is amenable for search. The problem specification, the rules of the application, implicitly define a graph. Following the terminology of [33], a problem space consists of states and transition functions to go from one state to another. For example, in chess a state would be a board description (piece locations, castling rights, etc.). The transition function implements the rules by which pieces move. In the traveling salesperson problem (TSP), a state can be a tour along all cities, or perhaps a partial tour. The transition function adds or replaces a city from a tour. In graph terminology, a state is called a vertex or node, and a transition is called an arc or edge.

The graph is treated as merely a formal representation of the problem, as yet devoid of meaning. It has not yet been decided what concepts like "payoff function" and "backup rule" mean. The problem graph is purely a syntactic description of the problem space. Semantics are added later. The graph has a number of interesting properties that can be exploited to improve the efficiency of the search. The following features characterize the properties of the basic graph. We assume that the graph is directed and finite.

Out Degree: The number of outgoing edges, or children, of a node is called its out degree or branching factor. The larger the out degree, the more difficult the search task. The 15-Puzzle has a branching factor that varies from 2 to 4. In chess, the average branching factor is roughly 40, although it can range from 0 to 100. An N-city TSP will have nodes with a branching factor of $N-1$. Search techniques may be able to exploit the distribution (absolute size or variability) of the out degree.

In Degree: The number of incoming edges, or parent nodes, is called the in degree of a node. Graphs with a constant in degree of 1 are called trees. Curiously, the problem spaces of most games studied by the field of game-*tree* search are not trees [38]. For example, in the game of chess many states can be reached via multiple move sequences, so-called transpositions. A search technique may be able to exploit the variability of the in degree.

Cycles: In addition to transpositions, graphs can contain cycles. A cycle is a situation where, via some path, a node is its own ancestor. The application rules allow cycles in, for example, chess and the 15-Puzzle, but not in the game of Go.

Graph Size: As noted above, we assume finite size. However, there is more to the size of the graph than the question of whether it is finite or not. In combination with other properties of the problem (such as time constraints, space constraints, and the computational cost of the transition function), the size of the problem graph influences the feasibility of finding an answer. The size of the search space, in part, determines the effectiveness of some search techniques.

5.2 Solution Definition

In this part of the problem solving process *meaning* is attached to some of the states. If the graph definition provides us with a syntactic description of the problem, then the solution definition associates semantics to the graph. The meaning of some states in the graph, mapped into a value, is defined by the application rules. For example, in chess all checkmate states have a known value. In the TSP, a tour that visits all cities and ends in the original one is a possible solution. The problem specification usually specifies a goal, such as checkmate the opponent or finding the shortest tour among a set of cities. The objective of the search is to find these goal or solution states, and to report back how they can be reached. Solutions are a subset of nodes in the search space, and solution density, solution depth, and the backup rule can be used to describe the relationship between these sets of nodes.

Solution Density: The distribution of solution states determines how hard searching for them will be. When there are many solution states it will be easier to find one, although determining whether it is a least cost solution (or some other optimality constraint) may be harder.

Solution Depth: An important element of how solution states are distributed in the search space is the *depth* at which they occur (the root of the graph is at depth 0). Search enhancements may take advantage of a particular distribution. For example, breadth-first search may be advantageous when there is a high variability in the depth to a solution.

Solution Backup Rule: The problem description defines how solution values should be propagated back to the root. Two-agent games use the minimax rule; optimization problems use minimization or maximization.

5.3 Resource Constraints

Resource constraints (space and time) permeate most of the design and implementation effort. They play a critical role in determining which enhancements are feasible. There are interesting trade-offs that can be made between the two [27, 19].

Space: Most algorithms used for solving real-world problems use storage to speed up their search. Space constraints (RAM or disk) may limit the amount of intermediate information that can be used by the search algorithm.

One could argue that with the steady decline in memory and disk prices, these space constraints are felt much less. While this is true, many algorithms are emerging that exploit the memory hierarchy to, for example, maximize cache usage. Algorithms should still be carefully designed to have a small working set to achieve best performance.

Time: In search, time is the single most important resource constraint. True, it can be traded off for space, but only up to a point. An unlimited amount of space is useless if one doesn't have the time to compute the data to fill it, and since the

problem graph is normally specified implicitly, it has to be built up during the search, which takes time that cannot be traded off.

Time constraints are the driving force for all the choices made during the different phases of the problem solving process.

5.4 Search Objectives

One of the most important decisions to be made is the objective of the search. This decision is influenced by the size of the problem graph, solution density and depth, and resource constraints. The choice of search objective defines a global stop condition. We distinguish two basic global stop conditions:

Optimization: The basic optimization question is to find the least cost solution to a problem. Optimization involves finding the best (optimal) value for the search problem. As soon as a solution has been found that is guaranteed to be optimal, the search stops (e.g. A*). (A small variation of this stop condition is to find all optimal solutions.) Given a problem graph, the properties that determine whether optimization is feasible are solution density, solution depth and in/out degrees.

Satisficing: Sometimes optimization is too expensive and one needs to get the best-quality answer possible subject to the resources available. This is the case in most two-agent problems where it is not possible to search to the end of the game, and in single-agent search where it is usually possible to quickly achieve a good solution, but considerably more expensive to find the optimal solution. In many other situations finding solutions is expensive, for example in many real-time situations, such as autonomous vehicle navigation. Recently there has been much interest in real-time algorithms that continue to find better solutions as their search time increases. Dean and Boddy have termed this group of algorithms anytime algorithms [13].

For satisficing searchers, a payoff, or evaluation function, is applied to each state encountered in the search. The evaluation function is a heuristic approximation of the true value of the state. The search progresses, trying to find the best approximation to the true solution, subject to the available resources.

The choice for global stop condition is influenced by computational efficiency—the faster the evaluation function the more states can be examined per time unit—and by the space and time that are available for the computation.

5.5 Domain Knowledge

The domain knowledge is at the heart of a high-performance search application, since the quality of the knowledge will significantly influence the efficiency of the search. At one extreme the application of perfect domain knowledge obviates the need for search. At the other extreme, a lack of domain knowledge will result in an ineffective (exhaustive) search.

Heuristic Evaluation Function: The heuristic evaluation function encodes application-dependent domain knowledge about the search. Typically, it is the most important component of a search application. Unfortunately, it has to be redeveloped anew for each problem domain. Since the function is application dependent, most of its internals cannot be discussed in a general way. The external characteristics, however, can.

There are many different types of information that can be returned by a heuristic evaluation. Some examples include: lower/upper bound estimates on the distance to solution (as is typically seen in most single-agent applications), point estimates on the quality of a state (as is typically seen in most two-agent applications), ranges of values (for example, B* [7]), and probability distributions (for example, BPIP [6]).

Backup Rule: The solution as defined in the problem specification typically gives a backup rule for propagating solutions back to the root of the search. This is often a simple minimizing or minimaxing operation. It is an abstract rule that is defined by the problem specification. In satisficing situations, however, the choice of heuristics used can change the semantics of the backup rule.

For example, in an AND/OR graph the cost of a solution could be backed up according to the MIN/SUM rule (for example, Conspiracy Numbers [30]). If the heuristic produces a scalar probability, then the backup rule will typically be product propagation. In some situations multiple values are returned, for example an upper and a lower bound, or the merit of a state and how much it costs to get there. Backing up a probability distribution requires even more elaborate processing.

Quality of Heuristics: The most important aspect of the heuristic evaluation function is the inherent error. For a given state s, the application designer wants to minimize $|h(s) - h*(s)|$, where $h(s)$ is the heuristic estimate of s's true value, and $h*(s)$ is the perfect-information value (if known). In general, the better the quality of $h(s)$, the more efficient the search. Typically, heuristic functions are good at estimating the true value of certain features, while they fail at others. The construction of good heuristic functions typically requires a lot of effort and ingenuity.

Satisficing and anytime algorithms are built on the premise that the error of search results diminishes with deeper searches. There has been some study into the pathological case when this assumption does not hold [32, 35]. Typically, real-world applications do not exhibit pathology (there are numerous studies showing this phenomenon [45, 9, 31]).

A special case that is of importance to minimization problems is where the heuristic function is admissible, that is, it never overestimates the cost of reaching a solution state. Its importance stems from the observation that the first solution found by, for example, iterative-deepening A* is guaranteed to be optimal if the heuristic function is admissible. These bounds can be used to prune irrelevant parts of the search space.

Parent/Child Correlation of Value: Related to the quality of the evaluation function is the *stability* of the values as one walks the graph. For many best-first and satisficing successor ordering techniques it is important that applying the heuristics to a parent node yields a value that is highly correlated to the backed up heuristic value of its children. Obviously, the better the general quality of the heuristics, that is, the smaller $|h(s) - h*(s)|$ is, the higher the correlation between parent and child values will be. (Although, in principle, also low quality evaluation functions can have a good heuristic stability, e.g. always returning 0.) An important special case is where the heuristic $h(s)$ is admissible and consistent.

Parent/Child Correlation of State: The graph-counterpart of the previous property, parent/child correlation of state, enables (or disables) the successful implementation of stable heuristics. The task of constructing heuristics that exhibit good stability becomes easier when few of the state features change in a transition. In many application domains one can choose a small number of relatively stable features in the state representation and use them to construct a good heuristic. The possible choices of a state representation can have a major impact on how easy it is to construct heuristics with a small error $|h(s) - h*(s)|$ that are computationally efficient, so that the part of the problem graph that can be searched within the time constraints is as large as possible. It often takes numerous design-and-test iterations before a satisfactory solution has been constructed.

Granularity of Values: The granularity of the heuristic function [46] is another issue that is partly constrained by properties of the search space, but where the designer of the heuristics has some degree of freedom.

The granularity of the heuristics can have a large influence on the choice of search techniques. For example, iterative deepening in single-agent search is a technique that searches for a solution whose cost does not exceed a threshold. If no solution is found, the threshold is incremented, and the search is restarted from the beginning. Its effectiveness is greatly affected by the granularity of the heuristic function [47]. In the 15-puzzle the heuristic (number of steps to a solution) is coarse grained, ranging from 0 to 64, with many nodes having the same value. In TSP the values returned by the heuristic function (length of a tour) are much more finely distributed, with many tours having a different cost. Iterative deepening works well with sliding-tile puzzles, but fails in TSP because of the high number of re-searches.

In satisficing the same phenomenon occurs with search techniques that iterate over the values returned by the heuristic. For example, most game-playing programs have migrated to manipulating integer values instead of the finer-grained floating point values.[4]

Search techniques exploit general properties of the search space in an application-independent fashion. Heuristics, on the other hand, are application-dependent search techniques. As stated before, the choice and implementation of heuristics provides in many ways the glue between the properties of the search space and

[4]Of course, a second argument is that integer arithmetic is faster than floating point calculations.

the search techniques. From a scientific viewpoint one would like to decouple application-specific heuristics from the more general search techniques. However, from an engineering viewpoint, trying everything conceivable to achieve the required performance for a certain problem, this separation can often seem absurd.

Next Node to Expand: The search algorithm together with heuristic information is used to decide on the next node to expand in the search. For some applications, the decision may be mechanical, such as depth-first, breadth-first or best-first, but heuristic information can be instrumental in ordering nodes from most- to least-likely to succeed. Search is graph traversal. The essence of what any combination of search algorithm/technique does is influence the choice of which node to expand next. The next node is influenced directly when descending the search tree, and indirectly when the decision is made to stop descending and backup to a parent node.

Select (down): When descending the graph one has to choose a child to expand next. This choice can have a large impact on the efficiency of the search. Breadth-first and depth-first are two basic strategies. Many inventive ways for choosing the next child to expand have been tried.

Local stop (up): Criteria to stop the search at certain nodes come in a multitude of flavors. Most search methods are designed not to expand the complete search space. Some techniques have been devised to decide when parts of the search graph do not have to be visited again. Many different pruning techniques have been proposed, based on search-space properties (pruning by domination, alpha-beta pruning), and on application-specific heuristics (forward pruning in chess).

Next, properties of the search space can be used to decide that a node temporarily will not be explored any deeper. Best-first node selection can be viewed as such a local stop criterion—stop when the children of a node are no longer the best. Many other search techniques have been devised, some based on properties of the search space, some on application-specific heuristics.

Having discussed the main properties of the search space, it is now time to turn our attention to the search techniques that are designed to take advantage of them. In the next section we switch from the vertical axis of Figure 1 to the horizontal axis.

6 Search Enhancements

This section classifies various search enhancements used. The enhancements have been grouped into classes, of which a few of the more interesting ones are discussed (the ones illustrated in Figure 1). Numerous enhancements have been classified, but are not included for brevity.

For each class, a representative technique was selected and its applicability to single-agent and two-agent search is discussed. The material is intended to be an illustrative sample, not exhaustive. Each technique is categorized by the preconditions of search domain properties that are necessary to use that technique. The effectiveness of some enhancements depends solely on the properties of the graph—not of the application—while others depend on the application and not on the graph. Since in most cases the preconditions necessary for using an enhancement are not tied to any fundamental property of an application, the search enhancements presented are applicable to a wide class of applications.

The example techniques are analyzed using a number of categories:

- Name. The commonly used name of the enhancement.
- Precondition. The conditions necessary for the enhancement to be applicable.
- Idea. A brief description of the idea behind the enhancement.
- Advantages. The benefits of using the enhancement (typically time and/or space).
- Disadvantages. The side effects of using the enhancement.
- Techniques. A brief summary of the idea as used in single-agent and two-agent search practice.

6.1 Eliminating Redundant States

Identical states can occur in a search. There are two ways this can happen. First, the graph may contain cycles. Second, path transpositions are possible (two independent paths through the graph reaching the same state). Ideally, the state should be searched once, and repeated occurrences of the state should reuse the previously computed information.

The presence of repeated states depends only on the application definition. Therefore the techniques for eliminating redundant states is independent of the algorithm selected.

Name: Cycle detection.
Precondition: In-degree is > 1. Two search paths can lead to the same state.
Idea: Repeated states encountered in the search need only be searched once. Search efficiency can (potentially) be improved dramatically by removing these redundant states.
Advantages: Reduces the search graph (tree) size.
Disadvantages: Increases the cost per node and/or storage required.
Techniques: The typical technique is to store positions in a hash table to allow for rapid determination if a state has been previously seen. This works for both cycles and path transpositions. An alternate technique that only detects cycles but uses little memory is to save the path used to reach a node and use it to check for a repeated state. Note that these techniques work for both single-agent [29] and two-agent [16] search. Finite state machines have been used to detect cycles in single-agent search [44].

6.2 State Space Enumeration

These techniques depend on the state space graph and on the definition of the solution space.
Name: Exhaustive search.
Precondition: Size of the state space graph and/or solution search tree be "small."
Idea: If the state space is "small" enough, then the entire graph can be examined and the optimal answer for each node computed. For some applications, traversal of the entire state space may not be necessary; one need only traverse the solution tree, ignoring parts of the state space that can logically be proven irrelevant.
Advantages: Optimal answer for some/all nodes in the state space.
Disadvantages: May require large amounts of time and/or space to traverse the state space and save the results.
Techniques: Several games and puzzles with large state spaces have been solved by enumeration (in conjunction with other enhancements), including single-agent applications (8-Puzzle [39] and the 12-Puzzle) and two-agent applications (Nine Men's Morris [14], Qubic [1] and Go Moku [1]).

The definition of "small" may be misleading. Nine Men's Morris has a state space size of O(10^{13}), while Go Moku has a state space size of over O(10^{100}), a seemingly impossibly large number. Solving a problem is a matter not only of the search-space size, but also the decision complexity [2].

6.3 Successor Ordering

The order in which the successors of an interior node are visited may effect the efficiency of the search.
Name: Move ordering.
Precondition: This enhancement is applicable if the order in which successor nodes are considered in can effect the size of the search tree.
Idea: Consider branches at an interior node in the order of most to least likely to achieve the best result.
Advantages: Successor ordering most benefits algorithms that use partial search results for additional cutoffs—the earlier good bounds are established the more of the remaining tree can be cut off (branch-and-bound and alpha-beta-based algorithms). In optimizing searches, searching the best move increases the likelihood that a (best) solution is encountered earlier in the search.
Disadvantages: Increased processing cost per interior node.
Techniques: There are many techniques for move ordering in the literature including the killer heuristic [43], previous best move ordering [43], iterative deepening, and the history heuristic [40]. Limited discrepancy search [18] is built around the notion of move ordering and relies crucially on its quality.

6.4 Iterative Refinement

Ideally, one should visit a state once, and only once. One of the major search results to come out of the work on computer chess was that repeatedly visiting a state, al-

though seemingly wasteful, may actually prove to be beneficial. Numerous iterative techniques have been used, including iterating on the search depth, search window, set of enhancements used, etc.
Name: Iterative deepening.
Precondition: Information from a shallow search satisfying condition d must provide some useful information for a more extensive search satisfying $d + \Delta$.
Idea: Search down a path until a condition d is met. If the entire tree has been searched with condition d, and no solution matching the search objective has been found and resources are not exhausted, then repeat a more extensive search to satisfy condition $d + \Delta$.
Advantages: Iterative deepening is primarily used because it reduces the space requirements of the application [24]. For searches under real-time constraints, iterative deepening facilitates time management, because it provides convenient places to stop the search with a reliable indicator of the quality of the search result (e.g. the search depth achieved). Results from previous iterations (stored in a transposition table) may improve move ordering and thus the potential for cutoffs.
Disadvantages: Repeated visitations cost time. The value of the information gathered must outweigh the cost of collecting it. In general, since the search trees grow exponentially, the cost of the early iterations is dwarfed by the cost of the last iteration.
Techniques: By storing the best move for each node searched, in each iteration the move ordering of another level of the search tree is improved [42, 43]. In optimizing searches, the algorithm usually iterates on a lower (upper) bound on the solution quality or search depth. The search either succeeds and a minimal (maximal) result has been found, or the search fails, in which case it is repeated with a larger (smaller) lower (upper) bound. Satisficing (resource-constraint) searches usually run until resources are exhausted and use the best result achieved so far. A related technique is recursive iterative deepening, which will not be discussed here.

6.5 Off-line Computations

It is becoming increasingly possible to pre-compute and store large amounts of interesting data about the search space that can be used dynamically at runtime. There are many well-known techniques, including pattern databases [12, 21] and opening books. The effectiveness of this technique depends ultimately on the heuristic evaluation function, although it works for a large class of applications.
Name: Solution databases.
Precondition: One must be able to identify goal nodes in the search (trivial).
Idea: The databases define a perimeter around the goal nodes. The search can stop when it reaches the perimeter. In effect, the database increases the set of goal nodes.
Advantages: The search can stop when it reaches the database perimeter.
Disadvantages: The databases may be costly to compute. Furthermore, the memory hierarchy (e.g., registers, level I cache, level II cache, RAM, and disk) makes random access to tables increasingly costly as their size grows.
Techniques: Solution (or endgame) databases have been built for a number of games, in some cases resulting in dramatic improvements in the search efficiency and in the quality of search result. In two-player games, the *Chinook* program has taken endgame

databases to the extreme, building a collection of 444 billion positions [41]. These databases are accessed at runtime, significantly reducing the search tree size and improving the search accuracy. In single-agent applications solution databases have been tried in the 15-Puzzle. An on-line version of this idea exists, dynamically building the databases at runtime (bi-directional [23] or perimeter search [28]).

6.6 Search Effort Distribution

The simplest search approach is to allocate equal effort (search depth) to all children of the root. Often there is application-dependent knowledge that allows the search to make a more-informed distribution of effort. Promising states can be allocated more effort, while less promising states would receive less. Extensive experimentation in two-player games shows that within a given time constraint, the quality of the search answer can be significantly improved by a judicious allocation of effort [4]. In two-agent search, numerous static methods have been used (see below) for adjusting the search effort (sometimes called forward pruning, selective search or selective deepening). Dynamic methods have proved more effective. Here search results are used to influence where the effort goes. Popular ideas used in practice include singular extensions [5], the null-move heuristic [15], and ProbCut [10].

For optimizing single-agent search, redistributing the search effort is of limited value since even if an extended search finds a solution, all possible non-extended nodes must still be checked for a better solution. However, if any solution is acceptable, then non-admissible heuristics can be used to extend/retract the search effort (both statically [26] and dynamically [20]). It is also beneficial for real-time single-agent search such as RTA* [25] and other anytime algorithms. Limited discrepancy search is a schema with the same intent: distribute the search effort in accordance with the move ordering heuristic [18].

Name: Static redistribution of search effort.
Precondition: Application-dependent knowledge can be a good predictor of the utility of extending/reducing the search.
Idea: Extend promising lines in the search; reduce the effort for subtrees that appear to have low potential. The decision is made based on static information about the current state.
Advantages: In optimizing search, this can increase the chances of discovering a (possibly non-optimal) solution quickly. In satisficing search, it can increase the reliability of the search result.
Disadvantages: In optimizing search, the discovery of a solution can be postponed. In satisficing search this might result in wasted search effort exploring low utility nodes.
Techniques: In single-agent search, numerous simplistic methods have been tried. For example, WIDA* (Weighted IDA*) scales the heuristic value by a constant (usually greater than 1) which has the effect of reducing the search more along lines with larger heuristic values [26]. In two-agent games, there are numerous application-dependent heuristics that have been used. For example, chess programs usually extend the search for checking or threatening moves, while lines where one side is down a lot of material are usually curtailed.

7 Analysis and Future Work

The previous sections illustrate that the same basic ideas for search enhancements have found their way into both single-agent and two-agent search and that even the algorithms are similar if not identical. It is often striking how closely related these enhancements are and how similar the algorithms behave when searching trees that were thought to belong to fundamentally different classes.

Even for moderately complex domains, the current state of the art requires a large programming, research and tuning effort to achieve high performance. Application programmers are required to re-implement general search algorithms and their enhancements over and over again. Even worse, the slow and agonizing process of debugging and tuning a multitude of interacting and counteracting search enhancements in search trees of millions, even billions, of nodes can be extremely time consuming and error prone.

The arguments outlined in this paper support the contention that the search space properties define the appropriate search algorithm/enhancements. The user defines the properties and then queries a catalogue of established techniques looking for those that match the properties of the application domain. This suggests that it should be possible to automate this process. One could construct a tool, a LEGO-box of search techniques, that, given a search-space description, automatically puts together a number of pre-fabricated pieces of template code and adapts them to the currently considered problem and its properties. Specifying the backup rule would be only one of many different properties to consider. Even though user-specified properties could be used in the beginning, this tool could ultimately detect certain properties itself and enable and disable, or even parameterize, specific enhancements accordingly [20].

Even though a generic solver likely would not execute as fast as a finely-tuned, custom built program, it could provide reasonable performance with the virtual guarantee of correctness. More importantly, a successful search enhancement can yield orders of magnitude in performance increase by dampening the exponent of the search-space complexity, whereas an excellent implementation can only save a small constant factor. Such a LEGO-program could be a benchmark for both speed and correctness for further, more specific implementations.

8 Conclusion

For decades researchers in the fields of single-agent and two-agent heuristic search have developed enhancements to the basic graph traversal algorithms. Historically the fields have developed these enhancements separately. Nilsson and Pearl popularized the AND/OR framework, which provided a unified formal basis, but also stressed the difference between OR and AND/OR graph traversal algorithms. The fields continued their relatively separate development.

This paper advances the view that the essence of heuristic search is not searching either single-agent or two-agent graphs, but which search enhancements one uses. First, the single/two-agent property is but one of the many properties of the search space that play a role in the design process of a high performance heuristic search application.

Second, the single/two-agent distinction is not the dominant factor in the design and implementation of a high-performance search application—search enhancements are. Third, most search enhancements are quite general; they can be used for many different applications, regardless of whether they are single-agent or two-agent.

The benefit of recognizing the crucial role played by search techniques is immediate: application developers will have a larger suite of search enhancements at their disposal; ideas first conceived of in two-agent search will not have to be rediscovered later independently for single-agent search, and vice versa. In an implementation the best combination of techniques depends on the expected search benefits versus the programming efforts, not on the single-agent or two-agent algorithm.

For twenty years, most of the research community has (explicitly and implicitly) treated single-agent and two-agent search as two different topics. Now it is time to take stock and recognize the pivotal role that search enhancements have come to play: the algorithm distinction is minor, and most research and implementation efforts are directed towards the enhancements. *All* the properties of the search space—not just the single/two-agent distinction—play their role in determining the effectiveness of that what heuristic search is all about: enhancing the basic search algorithms to achieve high performance.

9 Acknowledgments

This research was funded by the Natural Sciences and Engineering Research Council of Canada and iCORE.

References

[1] V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, 1994.

[2] V. Allis, H. van den Herik, and I. Herschberg. Which games will survive. In D. Levy and D. Beal, editors, *Heuristic Programming in Artificial Intelligence 2 – The Second Computer Olympiad*, pages 232–243. Ellis Horwood, 1991.

[3] S. Amarel. An approach to heuristic problem-solving and theorem proving in the propositional calculus. In J. Hart and S. Takasu, editors, *Systems and Computer Science*, 1967.

[4] T. Anantharaman. *A Statistical Study of Selective Min-Max Search in Computer Chess*. PhD thesis, Department of Computing Science, Carnegie Mellon University, 1991.

[5] T. Anantharaman, M. Campbell, and F. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, 1990.

[6] E. Baum and W. Smith. A Bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1–2):195–242, 1997.

[7] H. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[8] H. Berliner and C. McConnell. B* probability based search. *Artificial Intelligence*, 86(1):97–156, 1996.

[9] H.J. Berliner and C. Ebeling. Hitech. In T.A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 79–109, New York, 1990. Springer-Verlag.

[10] M. Buro. ProbCut: A powerful selective extension of the $\alpha\beta$ algorithm. *Journal of the International Computer Chess Association*, 18(2):71–81, 1995.

[11] J. Culberson and J. Schaeffer. Searching with pattern databases. In McCalla, editor, *Advances in Artificial Intelligence*, pages 402–416. Springer-Verlag, 1996. 11th Can. Soc. for Comp. Stud. of Intell. (CSCSI).

[12] J. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[13] T. Dean and M. Boddy. An analysis of time-dependent planning. In *AAAI*, pages 49–54, 1988.

[14] R. Gasser. *Efficiently harnessing computational resources for exhaustive search*. PhD thesis, ETH Zürich, 1995.

[15] G. Goetsch and M. Campbell. Experiments with the null move heuristic. In *AAAI Spring Symposium*, pages 14–18, 1988.

[16] R. Greenblatt, D. Eastlake, and S. Crocker. The Greenblatt chess program. In *Fall Joint Computer Conference*, volume 31, pages 801–810, 1967.

[17] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems, Science, Cybernetic*, SSC-4:100–107, 1968.

[18] W. Harvey and M. Ginsberg. Limited discrepancy search. In *AAAI*, pages 607–613, 1995.

[19] R. Holte and I. Hernadvolgyi. A space-time tradeoff for memory-based heuristics. In *AAAI*, pages 704–709, 1999.

[20] A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta, 1999.

[21] A. Junghanns and J. Schaeffer. Single-agent search in the presence of deadlock. In *AAAI*, pages 419–424, Madison/WI, USA, July 1998.

[22] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In *IJCAI-99*, pages 570–575, 1999.

[23] H. Kaindl, G. Kainz, A. Leeb, and H. Smetana. How to use limited memory in heuristic search. In *IJCAI-95*, pages 236–242, Montreal, 1995.

[24] R. Korf. Iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[25] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[26] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, July 1993.

[27] R. Korf and M. Reid. Complexity analysis of admissible heuristic search. In *AAAI*, pages 305–310, 1998.

[28] G. Manzini. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75:347–360, 1995.

[29] T. Marsland and A. Reinefeld. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, July 1994.

[30] D. McAllester. Conspiracy numbers for min-max searching. *Artificial Intelligence*, 35:287–310, 1988.

[31] P. Mysliwietz. *Konstruktion und Optimierung von Bewertungsfunktionen beim Schach*. PhD thesis, University of Paderborn, Paderborn, Germany, January 1994.

[32] D. Nau. Pathology on game trees revisited, and an alternative to minimaxing. *Artificial Intelligence*, 21:221–244, 1983.

[33] A. Newell and H. Simon. *Human Problem Solving*. Prentice-Hall, 1972.

[34] N. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

[35] J. Pearl. On the nature of pathology in game searching. *Artificial Intelligence*, 20:427–453, 1983.

[36] J. Pearl. *Heuristics*. Addison-Wesley, 1984.

[37] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1-2):255–293, November 1996.

[38] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Exploiting graph properties of game trees. In *AAAI*, pages 234–239, 1996.

[39] A. Reinefeld. Complete solution of the eight-puzzle and the benefit of node ordering in IDA*. In *IJCAI-93*, pages 248–253, Chambery, France, 1993.

[40] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):1203–1212, 1989.

[41] J. Schaeffer. *One Jump Ahead*. Springer-Verlag, 1997.

[42] J. Scott. A chess-playing program. In *Machine Intelligence 4*, pages 255–265, 1969.

[43] D. Slate and L. Atkin. Chess 4.5 — the Northwestern University chess program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118, New York, 1977. Springer-Verlag.

[44] L. Taylor and R. Korf. Pruning duplicate nodes in depth-first search. In *AAAI-93*, pages 756–761, 1993.

[45] K. Thompson. Computer chess strength. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56, Oxford, UK, 1982. Pergamon Press.

[46] W. Zhang and R. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79(2):241–292, 1996.

[47] W. Zhang and R. Korf. A study of complexity transitions on the asymmetric traveling salesman problem. *Artificial Intelligence*, 81(1-2):223–239, April 1996.